多线程 mini talk

多线程的意义

A 线程从键盘读入数据，B 线程要算一个计算量很大的任务，A 读入的数据和 B 的大部分计算过程无关。

- 单线程程序，先完成 A 后完成 B，但是用户一直不敲键盘，B 被拖累了。
- 多线程程序，A 和 B 分别是两个线程，A 在等用户输入的时候不会占用 CPU，这个时候正好可以算 B 中与 A 的输入无关的部分，提高了 CPU 利用率，加快了任务的完成。

多线程学习资源

- 实验和训练代码（和作业的需求贴近，启发性强）。
- 操作系统理论部分的"进程与线程"章节（系统了解理论知识）。
- Java语言提供的多线程类库与关键字，可以参考这个链接，或者《Java编程思想》和《Java核心技术》。
- 讨论区。
- 练手：LeetCode多线程题库。

完成作业所需必要知识点清单

- 如何**创建线程**：继承 `Thread` 类或者实现 `Runnable` 接口，并重写他们的 `void run()` 方法，该方法中描述的便是该线程启动起来之后要做的所有事情。
- 如何**启动线程**：调用 `start()` 方法。注意调用 `start()` 方法和直接调用 `run()` 方法的区别。
- `synchronized` 关键字，实现几个线程**互斥地**对同一个资源进行操作。
- `wait-notify(All)` 机制，实现几个线程之间的**协作**。
- 以上所有内容都可以在这里或者《Java编程思想》和《Java核心技术》上找到。

完成作业所需必要技能清单

> 从本单元同学们会发现，工具的学习同样重要。

- 多线程断点调试：可以选择任何一个没被阻塞的线程，让其单步执行，这意味着你可以去**手动调度这些线程的执行顺序**，从而找到自己的问题。一个视频，一个官方教程。

- 定时投放请求：电梯作业的请求是根据给定的时间戳输入的，在时间戳表示的时刻才会输入该请求。手动模拟定时输入不现实，一次性把所有请求都输进去会导致很多问题无法复现，所以可以学习一下定时发送方法。

可选学习内容

- 线程安全容器，如 `BlockingQueue`。学习链接
- 各种锁、信号量等高级/底层的机制。
- JProfiler，可以查多线程程序 CPU 使用情况，找到轮询的线程。使用方法
- 设计模式：生产者-消费者模式、状态模式、策略模式等。学习链接

多线程实践常见问题

> 多线程程序由于每次执行时线程被调度执行的顺序很可能不一样，所以以下错误均不保证每次运行能 100% 复现。

下面列举一下几类常见的问题、每类问题常见的问题代码、排查方案以及解决方案。

线程不安全

如果多个线程对同一个共享数据进行访问而不采取同步操作的话，那么操作的结果是不一致的。

比如创建 1000 个线程，每个线程都对 `object` 对象做一遍以下操作：

```
int val = object.getValue();
val++;
object.setValue(val);
```

很可能会出现线程 A 执行完前两条语句，还没来得及回写到 `object` 中，线程 B 便也开始执行了自己的第一条语句，导致 B 是对过时的值进行操作。最终查看 `object` 的值，大概率不是 1000。

这种问题排查时可以使用多线程断点构造多个线程同时访问某个资源的执行顺序；解决方案是做好互斥访问，具体实现上是使用 `synchronized` 或者线程安全类。

死锁

常见的情形是：线程 A 拿着资源 X，还想获得资源 Y；线程 B 拿着资源 Y，还想获得资源 X。A 和 B 互不相让，谁都没法继续运行下去。

```
// A 线程

synchronized(X) {
    // A 现在运行到这一行了
    synchronized(Y) {
        // balabala
    }
}

// B 线程

synchronized(Y) {
    // B 现在运行到这一行了
    synchronized(X) {
        // balabala
    }
}
```

另一种常见的情形：在其他线程还在 `wait` 的情况下，唯一还能跑的线程在用完公共资源之后，没有及时 `notify/notifyAll` 就润了（比如所有的电梯线程都在 `wait` 等待队列来新的需求，但是输入早就没有了，输入线程也结束了）。

出现死锁时，提交到评测机上有可能会出现 `REAL_TIME_LIMIT_EXCEEDED` 的错误。

这种问题的排查方法一般是多线程断点，构造最可能造成死锁的执行顺序。

轮询

CPU 时间过多地消耗在无意义的事情上，比如一直循环等待某个条件成立/不成立：

```
Queue<Integer> queue = new LinkedList<>();
while (true) {
    if (!queue.empty()) {
        queue.poll();
    }
}
```

还有一种常见的轮询原因：几个线程互相等待和唤醒，但是什么有意义的事情都没有做：

```
obj.notifyAll();
obj.wait();
```

在作业中写出类似的代码，在本地跑可能会导致电脑的风扇狂转。由于轮询是一种不良的 CPU 使用方式，消耗大量计算资源，所以轮询的程序提交到评测机上很可能会

`CPU_TIME_LIMIT_EXCEEDED`，表示你的程序 CPU 消耗太大。

正确的做法是当不满足条件时及时 `wait`，让出 CPU。